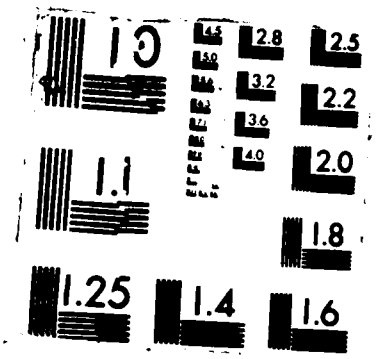
AD-A182 901 GRANULARITY ISSUES IN A KNOWLEDGE-BASED PROGRAMMING ENVIRONMENT..(U) CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.. P H FEILER ET AL. SEP 86 1/1

UNCLASSIFIED CMU/SEI-86-TM-11 ESD-TR-86-210 F/G 12/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

Technical Memorandum
SEI-86-TM-11

Carnegie-Mellon University
Software Engineering Institute

Granularity Issues in a Knowledge-Based Programming Environment

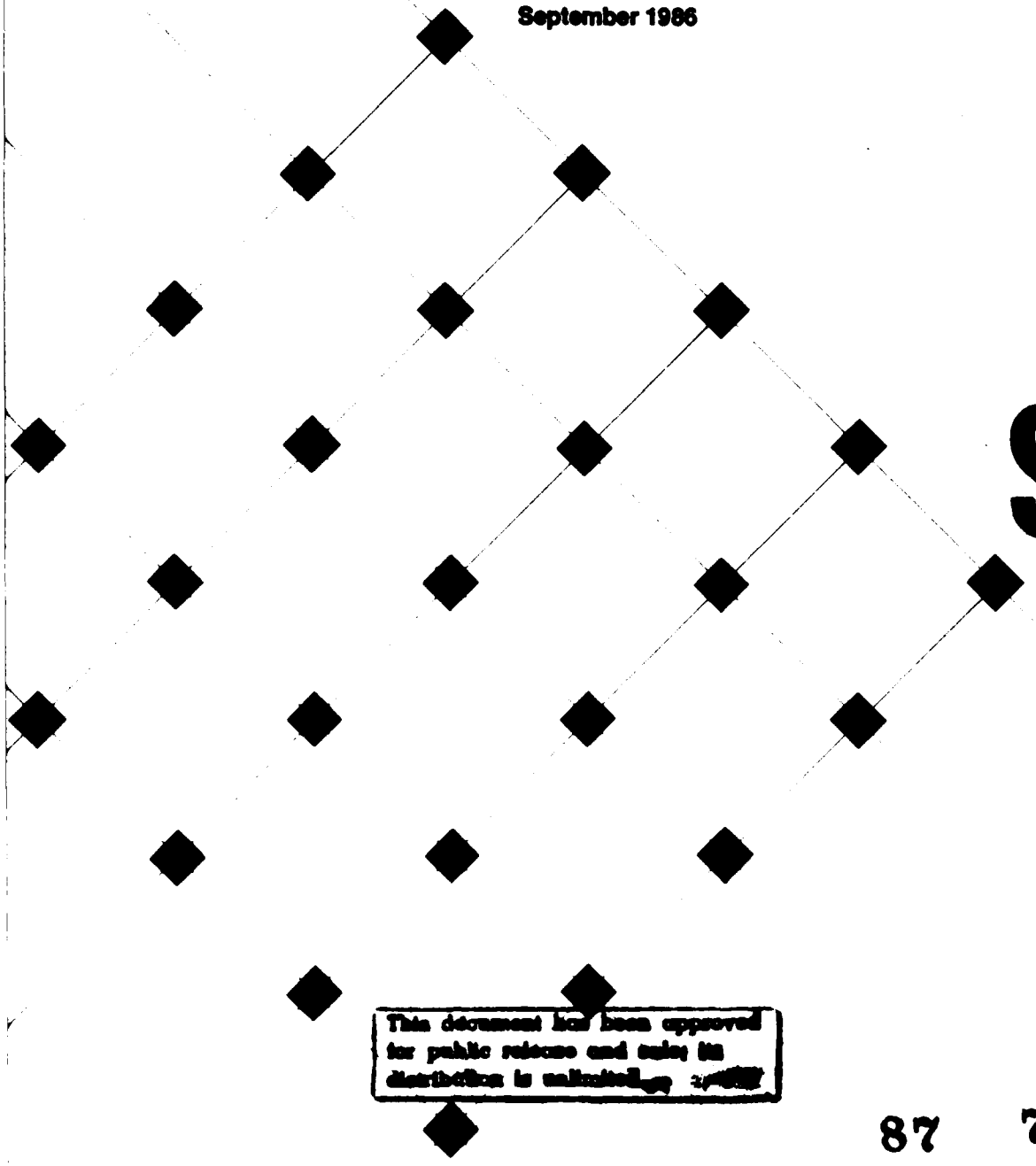
by

Gail E. Kaiser

and

Peter H. Feiler

September 1986



Technical Memorandum
SEI-86-TM-11
September 1986

Granularity Issues in a Knowledge-Based Programming Environment

by

Peter H. Feiler
Software Engineering Institute

and

Gail E. Kaiser*
Columbia University

*This paper was written while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA 15213.

The development and maintenance of SMILE is supported in part by the United States Army, Software Technology Development Division of CECOM COMM/ADP, Fort Monmouth, NJ and in part by ZTI-SOF of Siemens AG, Munich Germany.

Granularity Issues in a Knowledge-Based Programming Environment

Peter H. Feiler, Gail E. Kaiser[1]

ABSTRACT. We are in the process of developing MARVEL, a knowledge-based programming environment that assists multi-programmer software development teams in performing and coordinating their activities. During the design of MARVEL, we discovered that the granularity to which logical entities are refined in its software database and the granularity with which its software tools process the entities and report their results to the human users have a strong impact on the degree of intelligence that can be exhibited, as well as on the friendliness and performance of the environment. In this paper, we describe the many choices among alternative granularities and explain the decisions we made during the design of MARVEL.

1 Introduction

We are currently developing a knowledge-based programming environment called PROFESSORMARVEL, or MARVEL for short.[2] MARVEL is knowledge-based in the sense that it understands the logical entities and the activities involved in the software development process. It is an environment rather than a software tool because it actively participates in the software development process rather than remaining passive until explicit demands are made by its users. The primary functions of MARVEL are (1) to interactively answer queries about the current status of the development effort and the relationships among components of the target software system and (2) to automatically perform bookkeeping chores and simple development activities. This is in contrast to some other intelligent assistance systems such as the Programmer's Apprentice (also known as KBEmacs) [30], CHI (previously PSI) [24] and the Formalized System Development system (FSD) [2], which focus on the separate problem of automatic programming and program synthesis.

Unlike most other knowledge-based programming environments, MARVEL supports multi-programmer software development teams as well as individual programming efforts. For example, it includes facilities corresponding to Build [7] and SCCS [22] to coordinate simultaneous and sequential activities among multiple developers. However, MARVEL approaches these facilities in a participatory, knowledge-based fashion that enables it to automatically invoke the tools at the proper times without human intervention.

MARVEL is our second multi-user programming environment. Our first system, called SMILE [25, 9], was developed several years ago to support our research on the Gandalf project [20]. It

---

[1]The research presented in this paper was conducted while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

[2]Professor Marvel was the (Kansas) name of the man behind the curtain in the movie *The Wizard of Oz*.

has since been used extensively by other projects at Carnegie-Mellon University, at AT&T Bell Laboratories and at the University of Pisa, and has been distributed to approximately forty sites. SMILE was implemented in C and runs on Unix™. SMILE presents a fileless environment to its users, answers implicit and explicit queries, and automatically invokes various tools for its users. However, SMILE's knowledge is hardcoded into the environment and is not extensible; SMILE does not really 'understand' what it is doing.

Although we found SMILE very useful, and in fact relied on it for the implementation and maintenance of most of our research projects, we became convinced that an environment that did understand what it was doing could provide much more valuable assistance. Because of this, we have based our design of MARVEL on an architecture for intelligent assistance that consists of an objectbase and a model of the software development process. The *objectbase* maintains all software objects, including tools such as the editor and the compiler. The objectbase provides MARVEL with *insight* into the various classes of objects and the relationships among objects, such as one object is a component of another and a particular object may be applied to another object to produce a third object.

The *model* imposes a structure on programming activities. It consists of an extensible collection of production-like rules that specify the particular conditions that must exist for particular activities to be carried out. Metarules permit MARVEL to understand the rules and support *opportunistic processing*, where the environment can perform simple activities automatically, such as satisfying the preconditions of an activity and then carrying out the activity when it knows the results of the activity will soon be required by a user.

Insight and opportunistic processing are the topic of another paper [15], and will be discussed only briefly in this paper. The focus of this paper is the granularity issues that arose during our long experience with SMILE and during the subsequent design of MARVEL. In particular, we ran into several problems regarding the appropriate refinement of logical entities to be maintained as separate software objects and the units appropriate both for tools and for reporting the results of tool processing to the users. Choice of granularity affects the capabilities of the intelligent assistant, the friendliness *vs.* intrusiveness of the programming environment and, of course, performance and responsiveness. We believe that discussion of these issues, including an explanation of the decisions we made regarding MARVEL, will prove useful to other researchers who are in the process of building knowledge-based programming environments.

In the rest of this paper, we briefly explain the underlying basis for intelligent assistance utilized by MARVEL. We then address three areas of granularity issues: the granularity of structure in the objectbase, its impact on tools, and the granularity of processing automatically performed by the environment. We conclude by summarizing the significance of these issues for achieving intelligent assistance for software development.

2 A Basis for Intelligent Assistance

The distinguishing feature of an intelligent assistant is that it understands what it is doing [31]. We believe that both an objectbase and a model of the software development process are prerequisites to intelligent assistance. An assistant cannot understand why it performs particular activities unless it knows

- the properties of the objects it manipulates,
- the capabilities of certain objects (programmers and tools) to manipulate other objects,
- the preconditions required by each activity,
- the results or postconditions of each activity.

Therefore, MARVEL includes a general objectbase that maintains software entities and tools, and an extensible collection of rules that describe the preconditions and postconditions of software development activities.

2.1 The Objectbase

There are several possible forms for our objectbase to take. To maximize flexibility, we chose an objectbase similar to the objectbases of object-oriented programming languages, such as Loops [26]. In particular, we adopted their support for multiple inheritance and active values. The same concepts are found in the objectbases supported by other knowledge-based programming environments, such as the CommonLisp Framework (CLF) [3] and Refine™ [24].

In the MARVEL objectbase, each object is an instance of a class, which defines certain attributes of each object and inherits other attributes from its superclass(es). Some attributes define the relationships among the objects, while others trigger activities when they are accessed and/or updated. The software development activities applicable to the members of a class are defined as methods for the class.

MARVEL presents a 'fileless environment', exposing its users only to the logical structure of the target software system. As far as the users are concerned, the environment consists of a set of typed and interconnected software objects that represent both the system and its history of development. Object types include module, procedure, type, design description, user task (or development step), user manual, *etc.* Typing of these objects permits MARVEL to provide an object-oriented user interface similar to the Smalltalk-80™ environment [11]. This means that the environment makes available to each user only those commands that are relevant to the object under consideration.

The interconnections among software objects represent the logical structure of the system. The more detailed the structure, the more information is available for browsing and querying, and the more MARVEL can deduce which activities it can suitably perform and understand those tasks that the users carry out. These issues are addressed in Section 3.

2.2 The Model

There are also several possible forms for the rules we use to model the software development process. Again to maximize flexibility, we chose a style of rule developed in the program verification community. Each software activity is associated with preconditions and postconditions, as defined by Hoare [14]. The postconditions of an activity may satisfy the preconditions of future activities.

Our rules are similar to the production rules of Ops5 [5] and the automation rules of CLF [2] in that each rule has both a condition and action. When the condition is true, or satisfied, then the action may be carried out. Our rules are different from productions in that the action is divided into two parts, an activity and the postconditions. Because we have added postconditions to our rules, we refer to the original conditions as preconditions. We use the activity part of a rule to represent an integral software development task. For example, "compile module" is one activity and "edit procedure" is another. The preconditions of "compile module" are that the module is not compiled and that all its components have been analyzed without errors; there are two possible postconditions, exactly one of which is true: the module was compiled correctly and the compilation detected errors. Preconditions and postconditions are written as well-formed formulas (wffs) in the first order predicate calculus.

Our rules are maintained in a knowledgebase that also includes metarules, which permit MARVEL to perform activities and explain activities in terms of the rules. One metarule supports forward chaining: If the preconditions of a rule are satisfied, then MARVEL may perform the activity; the postconditions may satisfy the preconditions of other rules, which may then be applied. Another metarule defines backward chaining: If a user requests a particular activity, then MARVEL may attempt to satisfy its preconditions; this often requires the environment to first perform other activities whose postconditions match the preconditions of the original rule. MARVEL determines when to apply a particular metarule by employing strategies, which are collections of rules and metarules.

These and other metarules, together with the rules and the objectbase, support insight and opportunistic processing. One simple example of insight occurs when a user invokes the "change procedure" command, which enables editing of both the specification (header) and body of a particular procedure. MARVEL uses the results of its incremental analysis of dependencies among software components to inform the user of the potential consequences of this action; for example, each calling procedure will have to be modified if the number or order of the parameters are changed. A simple example of opportunistic processing occurs after a user completes the "change procedure" command by writing out the procedure from the editor. MARVEL notices that a postcondition of this activity is that the procedure is not analyzed, which is also the precondition of the "analyze procedure" activity; MARVEL uses forward chaining to automatically update its incremental analysis.

A full treatment of our objectbase and model and their application to insight and opportunistic processing is given in [15].

3 Granularity of Structure

The degree of intelligence that can be demonstrated by MARVEL, or any knowledge-based programming environment, is intimately tied to the granularity of independent entities maintained by its objectbase and to the granularity of the processing tools it has available. The granularity of structure determines to what extent the target software system is decomposed into separately stored entities. Separate storage may take the form of independent objects or of attributes of other objects. An appropriate choice of decomposition is influenced by several factors, including the need for separately recognizable logical entities, viewing and manipulation of the structure by the users, and space/time tradeoffs [19].

It is desirable to have the logical entities of the target system separately accessible for several reasons. In the first place, an object or attribute can be referenced from other parts of the system, while it is not possible to refer directly to only a portion of the information within an attribute. Examples of such entities are modules, procedures, macros, and global variables in the program domain, sections and subsections in the document domain, and plans, tasks, and developers in the management domain.

Logical entities depend on each other in various ways, and the interconnection structure must be accessible to maintain a consistent representation of the target system. These connections are viewed and manipulated both by the human users and by the environment. If the entities are accessible as separate objects, then the connections can be represented as relations among these objects. Examples include actual use dependency among software components such as procedures, macros and variables, intentional use dependency as expressed through export and import clauses of modules, compilation order, or referential use such as reference to a section or a citation in a document.

Status information is associated with particular logical entities. Status can represent the need to or the result of processing an entity, such as analysis of a procedure, code generation for a module, or running a section of a document through the document processor. It can also indicate coordination information among multiple users and between users and tools, *i.e.*, synchronization and version information [28]. MARVEL stores status information as attributes of objects.

The set of logical entities attributed with status information determines to some extent the degree of intelligence that can be built into the facilities provided by a knowledge-based programming environment. For example, to support 'smart recompilation' [29, 23], it is necessary to store status information about each intermodule symbol definition and use. This enables MARVEL to recompile only those entities that actually use modified symbols, as opposed to recompiling all modules that depend in any way on the module whose interface has changed. In contrast, SMILE supports import and export clauses, but does not permit more detailed relations among individual components, so it is not able to provide this kind of intelligent processing.

It may be desirable for users to navigate and manipulate the target system according to the structural units represented by logical entities. Syntax-directed editors [27] usually support cursor

movement and manipulation at the language construct level; Rational™ [1] takes this to an extreme by performing all processing in terms of the Diana [6] representation of Ada™ programs. Even text editors support a certain amount of structure, such as the electric-lisp mode in Emacs [13] recognizing matching parentheses. Graphical editors support manipulation of basic graphical symbols such as lines, circles and icons as well as composite graphical units. Word processing systems support character, word, sentence and paragraph manipulation. MARVEL supports viewing at the level of objects, using certain of their attributes as paths that can be followed by the browser to other objects.

User actions, especially modifications, have different effects for different logical entities. For example, editing of a comment does not affect analysis or code generation but is relevant to updating of hardcopy listings, whereas modification to a design specification not only affects other parts of the design, but also the implementation. Thus, MARVEL provides object-oriented commands to reflect these distinctions. MARVEL can then recognize a user's focus of attention as well as the extent of his or her modifications in order to be able to provide insight and to intelligently and effectively restore consistency of the target system. SMILE's more primitive user interface requires the user, for example, to give the "edit procedure" command to edit the body of a procedure and the distinct "change procedure" command to edit its specification (header).

4 Impact of Structure Granularity on Tools

The question arises as to the choice of the smallest logical entity that should be separately represented as an object. The answer must consider that there are two ways to have logical entities separately recognizable. One is by analyzing a 'larger' entity that is stored as a separate object in the objectbase, and deriving the 'smaller' entities as needed. The alternative is to represent each logical entity as a separate object. The appropriate choice is a tradeoff between the proliferation of objects and explosion of information to be stored on the one hand and reprocessing of information on the other. This manifests itself in a variety of ways.

Tools provided as part of a knowledge-based programming environment may require a special interface to the objectbase as they may not be able to cope directly with composite objects. For example, a compiler requires adaptation to be able to process modules as separate compilation units when they consist of sets of references to objects representing imported entities and a composition of objects representing the procedures, *etc.*, comprising the module. It is preferable to bring in existing tools without modifications; DSEE™ [18] does this for version control by providing a virtual interface between each tool and the version manager. We would like to do this in the general case, so MARVEL provides multiple *views* corresponding to the normal interfaces of the tools [10]. SMILE does not support views, and so is forced to store objects in the form expected by its tools; this sometimes results in duplication of information when the same kind of object is processed by multiple tools.

The status information of all components of a composite object may be accessed frequently. For example, the result of analysis of all components of a module must be positive before code

generation should be done. MARVEL can either compute the composite status value every time it is desired or it can cache the value in the module object and maintain it incrementally, as components are modified and re-analyzed, using, for example, finite differencing techniques [12].

Another example of status information is the error messages resulting from unsuccessful processing. Users tend to query them at times other than the time they are produced by the processing tool, except when the activity is performed on user demand. MARVEL stores error messages explicitly for strongly-typed languages, since recomputation is costly.

Interconnection information, such as actual use, may not require explicit representation, but this is orthogonal to whether the information is explicitly stored or dynamically determined. In other words, an attribute can be calculated only when needed, and information can be stored as part of some composite attribute rather than separately. One example of the first category is use dependency of local variables. This is rarely queried because all use sites are often displayed simultaneously. A user can visually search or use a viewer (editor) search command. Where explicit representation is necessary, the environment may explicitly store only the module in which an exported procedure is actually used (because the whole module has to be recompiled) and dynamically analyze the module when queried about the calling procedures, or it may explicitly store the procedure that calls the exported procedure but not every callsite within the procedure as those are easily found by direct search when needed. SMILE follows the former strategy, but the response time is widely variable and sometimes unacceptable, so MARVEL follows the latter course.

An objectbase permits tracking of modifications to objects. Representing logical entities as separate objects at appropriate granularities eliminates additional processing such as recognition of changes within regions of an object by the editor or content comparison algorithms (such as Diff [28]). For example, procedures may be represented as composite objects consisting of the specification, a description, and the implementation. Using this as the lowest level structural granularity permits MARVEL to limit side-effect propagation considerably, since other entities can be affected only when the specification is changed.

Decomposition of the target system into separate objects at a small grain places certain requirements on the object-viewing and browsing capabilities of the environment. On one hand, a user should not be starved of information as is the case with single-level object viewers. For example, viewing a module may result in display of only the names of components, without even an indication of their type. More context should be provided to the users.

On the other hand, a user should not be overloaded with information, which may lead to disorientation and confusion. An example of this is the presentation of the target system as a single textual unit, decorated with all available status information — possibly encoded in a range of symbols. A balance must be struck as to the amount of information to be displayed at any time and the desire to reduce explicit querying for information. This may change over time as the users carry out different activities. For example, while making major changes to the system a user has little interest in code generation status. Similarly, when examining an imported module, the

user's view should be limited to its specification. SMILE solves these problems with distinct commands for different levels of detail, while MARVEL includes an objectbase viewing and browsing capability supporting multiple views and multi-level viewing, which makes best use of available screen space through multiple viewing panes.

5 Granularity of Processing

As previously discussed, the tools as well as the users can take advantage of multiple views. A related issue is whether or not the users should have multiple 'views' of the tools. The granularity of processing determines the responsiveness of the environment as well as the intelligence perceived by its users. Responsiveness refers both to feedback to the users regarding inconsistencies in the target system and to processing of the target system to derive other representations, *e.g.*, to prepare for execution or for formated printing. MARVEL and other knowledge-based programming environments are interactive environments that attempt to increase user productivity. This means that each user should get feedback while in the context of the problem spot; the environment displays intelligence by understanding the user's notion of context and relating it to the results of the tools.

This also means that a user should not have to wait excessively for the computer to complete its share of the work. This is accomplished by processing at the appropriate level of granularity and by processing opportunistically. The availability of both forward and backward chaining permits MARVEL to perform activities any time between when the preconditions are satisfied and when the postconditions are required. Furthermore, not all processing has to be done at the same level of granularity. Granularity of processing that results in feedback to the user is strongly influenced by the context and time in which feedback is expected. Note that feedback may involve simple visual cues, such as changing the font of the prompt, rather than immediately dumping all the error messages on the user's display.

Granularity of processing resulting mainly in derived entities such as object code is primarily influenced by the following tradeoff. On the one hand, we have the possibility of processing many small units, thus reducing the time of processing one unit, yet causing possibly redundant processing of the same unit when it is frequently affected by changes to other small units. On the other hand, processing larger units at less frequent intervals leads to the expense of longer delays when the users need the results. In some cases, this problem means exploration of separate processing techniques in order to avoid processing of the complete target system. Examples include linking in pieces through use of indirect references [8], and formatting in pieces by a document processor through maintenance of formatting context — as is supported by Scribe [21].

Feedback to the users can occur at several levels of granularity, where the grain size chosen may be different for different kinds of analysis. One form of feedback is enforcement of a particular kind of consistency. The most prominent example is enforcement of syntactic consistency, as done by syntax-directed or form editors. This is accomplished either by limiting the choices of

entry to acceptable ones, *e.g.*, by providing a menu with the legal set of constructs, or by immediately checking and rejecting invalid entries. Another form of feedback is checking for consistency and reporting any violations, but accepting the input into the objectbase. In this case, MARVEL records whether or not objects have been checked for each kind of consistency and, if so, the results of each analysis.

During different phases of development and maintenance, a user may desire feedback for the same kind of consistency at different granularity levels. For example, while writing a new piece of code, a user does not want to be told repeatedly about the use of an undefined identifier until he has completed his activity with the procedure or module. However, when carrying out minor corrections, more immediate feedback is desirable. MARVEL offers such flexibility by separating checking from reporting. In this way, checking for a particular kind of consistency is always performed at the same level of granularity — the smallest level for which feedback is desired — with one set of analysis processes. Reporting can be realized by querying the results of checking, and MARVEL does this by performing queries automatically at different times as determined by the reporting strategy.

This behavior is in contrast to SMILE, which initially performed compilation at the level of procedures and immediately informed the user of any errors. We found this behavior unacceptable. SMILE was modified to perform symbol resolution and type checking at the procedure level, but to apply compilation only to modules. Errors were no longer reported except in response to user queries, but their detection was indicated by an unintrusive visual change in the display that remained until the errors were corrected.

6 Conclusions

The fundamental tradeoffs regarding granularity of logical entities and of automatic processing demonstrate the impact of the choices of granularity on the apparent intelligence of an environment as well as on its responsiveness and performance. The most notable choices we made for PROFESSORMARVEL are notably

- A knowledge-based programming environment can more quickly answer more complex queries when it incrementally updates its analysis of the relationships among logical entities of the target software system and also maintains these entities refined to the level of relationships among individual software components rather than among modules;
- An environment is less intrusive and more informative when the granularity of automatic processing is separated from the granularity for automatic reporting of the results of processing, and it is not difficult to separate these behaviors for semantic analysis, compilation and other activities.

We believe that these choices are also appropriate for most other knowledge-based programming environments. While SMILE was targeted for C, we designed MARVEL to support either C, CommonLisp, or Ada. We also kept in mind document formatting languages such as Scribe and project management facilities such as those found in CMS [16] and DSEE [17]. Thus our conclusions cover a wide range of possible entities as well as tools.

REFERENCES

[1] James E. Archer, Jr. and Michael T. Devlin.
Rational's Experience Using Ada for Very Large Systems.
In *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*, pages B.2.5.1-B.2.5.11. Houston, TX, June, 1986.

[2] Robert Balzer.
A 15 Year Perspective on Automatic Programming.
IEEE Transactions on Software Engineering SE-11(11):1257-1268, November, 1985.

[3] Robert M. Balzer.
Living in the Next Generation Operating System.
In *Proceedings of the 10th World Computer Congress (IFIP Congress '86)*. Dublin, Ireland, September, 1986.
To appear.

[4] David R. Barstow, Howard E. Shrobe and Erik Sandewall.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984.

[5] Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin.
Programming Expert Systems in OPS5.
Addison-Wesley Publishing Co., Reading, MA, 1985.

[6] *Diana Reference Manual*
Carnegie-Mellon University, Department of Computer Science, 1981.

[7] V.B. Erickson and J.F. Pellegrin.
Build — A Software Construction Tool.
AT&T Bell Laboratories Technical Journal 63(6):1049-1059, July-August, 1984.

[8] Peter H. Feiler and Raul Medina-Mora.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5):472-482, September, 1981.

[9] Peter H. Feiler and Gail E. Kaiser.
Intelligent Assistance without Artificial Intelligence.
Submitted for publication.

[10] David Garlan.
Views for Tools in Integrated Environments.
In *Proceedings of the IFIP WG 2.4 International Workshop on Advanced Programming Environments*. June, 1986.
To appear as a book published by Springer-Verlag.

[11] Adele Goldberg.
The Influence of an Object-Oriented Language on the Programming Environment.
In *Proceedings of the 1983 ACM Computer Science Conference*. February, 1983.
Reprinted in [4].

[12] Allen T. Goldberg.
Knowledge-Based Programming: A Survey of Program Design and Construction Techniques.
IEEE Transactions on Software Engineering SE-12(7):752-768, July, 1986.

[13] James A. Gosling.
Unix Emacs
Carnegie-Mellon University, Department of Computer Science, 1982.

[14] C.A.R. Hoare.
An Axiomatic Approach to Computer Programming.
Communications of the ACM 12(10):576-580, 583, October, 1969.

[15] Gail E. Kaiser and Peter H. Feiler.
Intelligent Assistance for Software Development and Maintenance.
In progress.

[16] Beverly L. Kedzierski.
Knowledge-Based Project Management and Communication Support in a System Development Environment.
In *Proceedings of the 4th Jerusalem Conference on Information Technology.* Jerusalem, Israel, May, 1984.

[17] David B. Leblang and Robert P. Chase, Jr.
Computer-Aided Software Engineering in a Distributed Workstation Environment.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104-112. Pittsburgh, PA, April, 1984.

[18] David B. Leblang and Gordon D. McLean, Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *GTE Workshop on Software Engineering Environments for Programming in the Large*, pages 122-127. June, 1985.

[19] John R. Nestor.
Toward a Persistent Object Base.
In *Proceedings of the IFIP WG 2.4 International Workshop on Advanced Programming Environments.* June, 1986.
To appear as a book published by Springer-Verlag.

[20] David Notkin.
The GANDALF Project.
The Journal of Systems and Software 5(2):91-105, May, 1985.

[21] Brian K. Reid and Janet H. Walker.
SCRIBE Introductory User's Manual
Carnegie-Mellon University, Department of Computer Science, 1979.
2nd edition.

[22] M. J. Rochkind.
The Source Code Control System.
IEEE Transactions on Software Engineering SE-1:364-370, 1975.

[23] Robert W. Schwanke and Gail E. Kaiser.
Version Inconsistency in Large Systems.
Technical Report RTL-86-TR-072, Siemens Research and Technology Laboratories, April, 1986.

[24] Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold.
Research on Knowledge-Based Software Environments at Kestrel Institute.
IEEE Transactions on Software Engineering SE-11(11):1278-1295, November, 1985.

[25] Barbara J. Staudt, Charles W. Krueger, A.N. Habermann and Vincenzo Ambriola.
The GANDALF System Reference Manuals.
Technical Report CMU-CS-86-130, Carnegie-Mellon University, Department of Computer Science, May, 1986.

[26] Mark Stefik and Daniel G. Bobrow.
Object-Oriented Programming: Themes and Variations.
AI Magazine 6(4):40-62, Winter, 1986.

[27] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM 24(9), September, 1981.
Reprinted in [4].

[28] Walter F. Tichy.
RCS — A System for Version Control.
Software — Practice and Experience 15(7):637-654, July, 1985.

[29] Walter F. Tichy.
Smart Recompilation.
ACM Transactions on Programming Languages and Systems (TOPLAS) 8(3):273-291, July, 1986.

[30] Richard C. Waters.
KBEmacs: Where's the AI?
The AI Magazine VII(1):47-56, Spring, 1986.

[31] Terry Winograd.
Breaking the Complexity Barrier (Again).
In *Proceedings of the ACM SIGPLAN-SIGIR Interface Meeting on Programming Languages — Information Retrieval*, pages 13-30. Gaithersburg, MD, November, 1973.
Reprinted in [4].

END

9-87

DTIC